Análisis de la implementación software de un conformador de señales ultrasónicas para tiempo real

D. Romero-Laorden , J. Villazón-Terrazas , M. Santos Peñas , M. A. García-Izquierdo , O. Martínez-Graullera

Resumen

Este trabajo analiza la implementación software en un sistema de imagen ultrasónica del Total Focusing Method para la compensación dinámica en tiempo real de los tiempos de vuelo para emisión y recepción de todos los puntos de la imagen. Para ello, haciendo uso de técnicas GPGPU, se analizan dos diferentes alternativas de implementación, mostrando como una planificación adecuada de acceso a los datos permite mejorar los tiempos de ejecución del algoritmo.

Palabras Clave: Imagen Ultrasónica, Procesamiento de Señal, Computación Paralela, GPU, CUDA

1. Introducción

En los últimos años se ha popularizado el uso de plataformas *Graphics Processor Unit* (GPU) como base para el desarrollo de sistemas de computación paralela, *General Purpouse Graphics Processor Unit* (GPGPU). Su bajo coste, sus altas prestaciones y el desarrollo de herramientas específicas para este tipo de plataformas, han abierto la posibilidad de plantear el uso de la GPU como sustitutivo total o parcial de sofisticado hardware específico para procesamiento en tiempo real.

La imagen ultrasónica es una de las técnicas más populares para la visualización del interior de cuerpos opacos. Su bajo coste, sencillez y eficiencia, han hecho posible que su uso se extienda de forma significativa tanto en el campo del diagnóstico médico como en el de la Evaluación No Destructiva (END). Esta técnica se basa principalmente en el análisis de los ecos que se producen por las discontinuidades en el material de un cuerpo, en respuesta al paso de una perturbación mecánica inducida artificialmente. De este análisis se desvela la estructura interna del objeto bajo estudio y la presencia de anomalías en la misma.

Dentro de este campo, es habitual hacer uso de sistemas multitransductor que capturan miles de señales desde diferentes posiciones de referencia y que analizadas en procesos de conformación pueden componer una imagen de la región insonificada. En este sentido el diseño de conformadores de haz, que involucra etapas intensivas de procesamiento paralelo de datos, ofrece un ejemplo paradigmático para la aplicación de las nuevas técnicas de procesamiento paralelo en GPU. Un conformador de haz, es un sofisticado filtro espacio-temporal con múltiples señales de entrada y alto paralelismo, cuya función es componer la imagen y cuya implementación es tradicionalmente hardware (Parrilla, 2004; Camacho, 2010).

Podemos optar entre muchas formas de composición de imagen en función de cómo se realice la insonificación, la adquisición, o cómo se combinen las señales recogidas (Szabo, 2004). El problema clásico de conformación de haces en recepción y tiempo real para un array de N elementos e I puntos en la imagen tiene un nivel de complejidad $O(N \times I)$, y a día de hoy se han desarrollado distintos tipos de soluciones tanto a nivel de hardware (Siritan et al., 2013; Wall and Lockwood, 2005; Camacho et al., 2007), como de software (So et al., 2011).

En lo que respecta al software, los primeros trabajos de paralelización en CPU datan de principios del siglo XXI, con el incremento de la popularidad de los *clusters* computacionales de bajo coste (Zhang et al., 2002; Kortbek et al., 2007). Estas contribuciones presentaban soluciones interesantes, pero con un éxito reducido a la hora de obtener tasas de velocidad que se puedan considerar de tiempo real (25 imágenes/segundo). En esta línea en (Hansen et al., 2011) se presenta una solución para la conformación de haz en recepción ya orientada a arquitecturas multi-core, que con 8 cores es capaz de generar una imagen con 300000 píxeles en 142, 3 segundos para un array de 192 sensores. En paralelo a este trabajo, otros autores se enfocan a la búsqueda de soluciones sobre GPUs (Nilsen and Hafizovic, 2009; Wang et al., 2011), obteniendo ahora ya tasas de imagen

aceptables de acuerdo con los parámetros de la aplicación. A partir de ahí la diversidad de plataformas ha permitido algunos estudios comparativos entre CPU, GPU y FPGA (Birk et al., 2011) que, aunque hacen énfasis en el buen funcionamiento del hardware dedicado, no han supuesto un freno al desarrollo de soluciones software cada vez mejores y sobre algoritmos de conformación de haz más sofisticados y complejos. Siguiendo esta línea, este trabajo se centra en las técnicas de Apertura Sintética (SAFT).

Partiendo de la adquisición independiente de las señales correspondientes a cada par de emisor-receptor se puede evaluar la contribución de cada señal sobre cada punto de la imagen, generando así una imagen de muy alta calidad. Ésta técnica se conoce como *Total Focussing Method* (TFM) donde su rango dinámico viene determinado por el número de señales involucradas y la resolución lateral es la máxima que la apertura es capaz de ofrecer en cada punto de la imagen (Holmes et al., 2005). Su importancia radica entonces, en que esta técnica se considera un *gold-standard* de calidad (Martín-Arguedas, 2010; Jensen et al., 2005; Camacho, 2010). No obstante, el coste computacional de esta operación es superior al del problema clásico de conformación de haz en recepción. Así, dado un array de N elementos, si consideramos el conjunto total de las $N \times N$ señales independientes, conocido como *Full Matrix Capture* (FMC), la complejidad del algoritmo TFM+FMC es $O(N \times N \times I)$.

El problema no es nuevo y ha sido objeto de interés desde principios de siglo para los desarrolladores de sistemas. La mayoría de estos trabajos han seguido la línea de desarrollar hardware específico basado en FPGAs, produciendo así voluminosos equipos de laboratorio (Nikolov, 2001; Jensen et al., 2005). Sin embargo, no es hasta hace muy poco cuando esta línea se ha materializado en el desarrollo de un producto comercial (M2M, 2015). Este producto es capaz de generar un TFM en tiempo real para un array de 64 sensores (4096 señales), consiguiendo para una imagen de 256×256 píxeles una tasa de 25 imágenes por segundo.

En la literatura el tratamiento del problema de conformación de haces sobre plataformas software se ha realizado principalmente como etapa de post-procesamiento y, en consecuencia, no se han desarrollado apenas implementaciones orientadas al tiempo real. Los primeros trabajos que abordaron este problema para su aplicación en tiempo real sobre GPGPU fueron publicados en 2009 (Romero-Laorden et al., 2009). En este trabajo se hacía uso del *Minumum Redundancy Coarray* (MRC) (Martín-Arguedas, 2010) para reducir la complejidad a $O(N \times I)$ y obtener así una tasa de 30 imágenes por segundo para un tamaño de imagen de 256×256 píxeles y 128 sensores. En trabajos posteriores, profundizando en este concepto (TFM+MRC) y con la mejora de las plataformas de cálculo, las tasas de imagen llegaron a superar las 150 imágenes por segundo (Romero-Laorden et al., 2011, 2012).

Respecto al problema que nos ocupa, TFM+FMC, la bibliografía sólo presenta un único trabajo que desarrolla un sistema para tiempo real (Sutcliffe et al., 2012). Este sistema, desarrollado para END y dotado de una conexión con el equipo de adquisición por ethernet, obtiene una tasa de 8 img/sec para una imagen de 120×120 y 64 elementos operando sobre una NVIDIA Fermi con 384 cores. Más recientemente se ha publicado un estudio comparativo que incluye tanto CPU como GPUs (Xeon X5690 con 6 cores, GPU NVIDIA Fermi con 512 y 448 cores, y GPU ATI 1536 cores), obteniendo una tasa de imagen máxima de 5 img/sec para una imagen de 200×200 píxeles (Rougeron et al., 2013).

El presente trabajo, parte de la experiencia desarrollada para TFM+MRC en (Romero-Laorden et al., 2011) para analizar la capacidad de las unidades de procesamiento gráfico para realizar TFM+FMC en tiempo real. La experiencia previa reveló la importancia del uso de los recursos de memoria y del planteamiento algorítmico para conseguir el rendimiento deseado. Así, partiendo de la solución directa se proponen varias implementaciones y se analizan los factores que determinan su eficiencia. Los resultados revelan que trabajando sobre un escenario industrial para arrays de 64 y 128 elementos es posible obtener altas tasas de imagen.

La estructura del artículo es la siguiente. Tras la introducción, en la Sección 2 se presenta el algoritmo de conformación de haz que se va a utilizar. En la Sección 3 se describe con detalle la implementación en GPU de este algoritmo, tanto directa como optimizada. La Sección 4 evalúa el rendimiento de las distintas propuestas para diversas plataformas y finalmente se presentan las conclusiones.

2. El algoritmo de conformación de haz

Como se muestra en la Figura 1, suponemos un array lineal de N elementos situados sobre el eje X y una región de interés sobre el plano (x, z) que ha sido discretizada en el espacio $(x[k,l])$. La aplicación del TFM sobre el conjunto total de señales adquiridas es capaz de generar una imagen $a[k,l]$, donde cada punto tiene el valor:

$$a[k,l] = \sum_{i=1}^{N} \sum_{j=1}^{N} b_{ij} s_{i,j}(t) \cdot \delta\left(t - t_{i,j}[k,l]\right) \tag{1}$$

siendo $s_{i,j}(t)$ la señal correspondiente al emisor e_i y al receptor e_j; b_{ij} es el coeficiente correspondiente del filtro espacial que constituye el conformador, y $t_{i,j}[k,l]$ es el tiempo de propagación desde el emisor e_i hasta el punto de la imagen $x[k,l]$ más el tiempo de propagación desde este punto al receptor e_j (2).

$$t_{i,j}[k,l] = t_i[k,l] + t_j[k,l] - t_0 = \frac{|x_i - x[k,l]| + |x_j - x[k,l]|}{c} - t_0 \tag{2}$$

donde c es la velocidad del propagación del sonido en el medio, x_i y x_j las localizaciones de los elementos e_i y e_j sobre el plano XZ, y t_0 el tiempo inicial de espera para la adquisición. Debido al muestreo temporal del sistema de adquisición, la señal recibida debe ser expresada como:

$$s[n] = s(t) \cdot \delta(t - n\tau_s) \tag{3}$$

donde $1/\tau_s$ es la frecuencia de muestreo. Para ajustar el tiempo de vuelo $t_{i,j}[k,l]$ sobre el espacio de muestreo necesitamos transformarlo en un valor de índice de muestra $m[i,j,k,l]$:

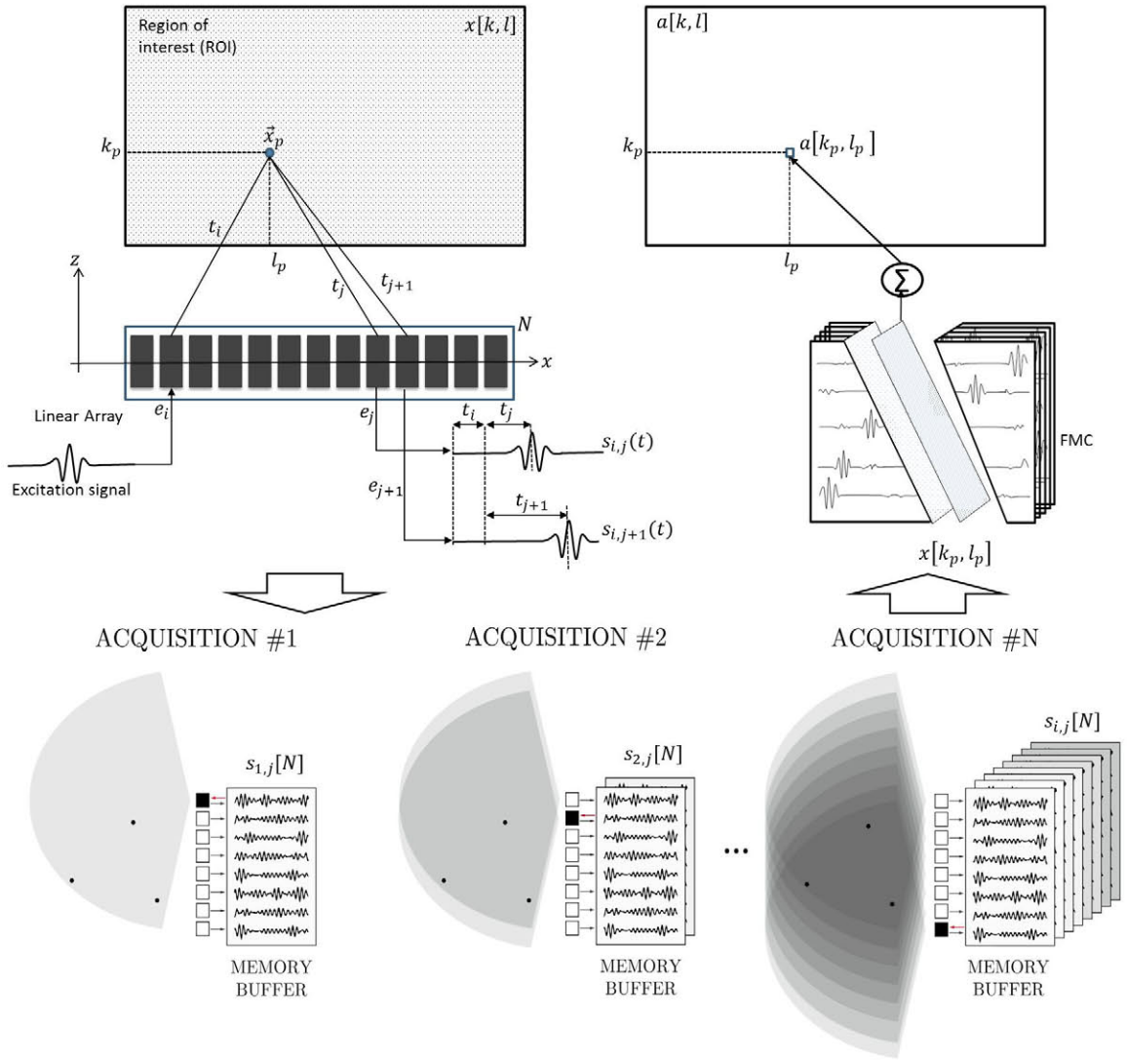


Figura 1: Diagrama que ilustra el proceso de adquisición FMC. Se presenta un array lineal de N elementos, un reflector $\vec{x}_p$ y el tiempo de vuelo hacia varios elementos del array e_i, e_j y e_{j+1} desde el mismo. A cada elemento del array se le hace corresponder un disparo, y por cada disparo se capturan las N señales correspondientes a los N elementos del array con M muestras. Sobre el volumen total de muestras capturadas para un punto $a[k_p, l_p]$ se determina el subconjunto de muestras que compensa los tiempos de propagación hasta ese punto desde cada par emisor-receptor

$$m[i,j,k,l] = floor\left(\frac{t_{i,j}[k,l]}{\tau_s}\right) \tag{4}$$

$$\Delta m[i,j,k,l] = \frac{t_{i,j}[k,l]}{\tau_s} - floor\left(\frac{t_{i,j}[k,l]}{\tau_s}\right) \tag{5}$$

siendo $\Delta m[i,j,k,l]$ un coeficiente que podemos usar para ajustar el retardo entre dos muestras consecutivas por medio de un interpolador lineal. De tal forma que la muestra deseada queda expresada como:

$$\S_{i,j}[k,l] = s_{i,j}[m[i,j,k,l]] \quad + \\ \Delta m[i,j,k,l]\left(s_{i,j}[m[i,j,k,l]+1] - s_{i,j}[m[i,j,k,l]]\right) \tag{6}$$

De este modo, el TFM puede expresarse como:

$$a[k,l] = \sum_{i=1}^{N}\sum_{j=1}^{N} b_{i,j}\S_{i,j}[k,l] \tag{7}$$

Dado que las señales $s_{ij}(t)$ están en radiofrecuencia, la imagen $a[k,l]$ se obtiene en radiofrecuencia. Con el objetivo de suavizar la imagen y hacerla adecuada para su visualización es necesario calcular su envolvente. Este proceso suele realizarse mediante la transformada de Hilbert (Oppenheim et al., 1989) que permite formular $a[k,l]$ analíticamente como (7), donde $a_I[k,l]$ y $a_Q[k,l]$ son los componentes de fase y cuadratura respectivamente:

$$a[k,l] = a_I[k,l] + ja_Q[k,l] \tag{8}$$

Así, la envolvente puede ser calculada como:

$$A[k,l] = \sqrt{a_I^2[k,l] + a_Q^2[k,l]} \tag{9}$$

Sin embargo, debido a la condición de periodicidad que implica la FFT, la transformada de Hilbert introduce en los bordes de la imagen artefactos que la distorsionan. Con el fin de evitarlos, la descomposición de la señal analítica puede hacerse sobre las señales adquiridas $s_{i,j}[n]$ en lugar de sobre la imagen. Donde las señales se expresan en su forma analítica por sus componentes de fase y cuadratura:

$$s_{i,j}[n] \rightarrow s_{Ii,j}[n] + j s_{Qi,j}[n] \tag{10}$$

De esta manera los artefactos se producen sobre los extremos de las señales, que pueden quedar fuera del campo de interés de la imagen y que en cualquier caso pueden ser filtrados por el propio proceso de *beamforming*.

El inconveniente de esta solución es que ahora se necesitan dos cadenas de procesamiento, una para la componente de fase y otra para la componente de cuadratura, lo que incrementa la complejidad del proceso y los requerimientos de memoria. Con objeto de reducir este coste podemos realizar una etapa previa de preprocesamiento conocida como reducción *half-matrix* (Holmes et al., 2008; Hunter et al., 2008; Sutcliffe et al., 2012) que establece que dada la equivalencia entre los tiempos $t_{i,j}[k,l]$ y $t_{j,i}[k,l]$, las señales $s_{i,j}[n]$ y $s_{j,i}[n]$ pueden ser procesadas de forma conjunta mediante una nueva señal, $\hat{s}_{i,j}[n]$, que es calculada por:

$$\hat{s}_{i,j}[n] = s_{i,j}[n] + s_{j,i}[n] \tag{11}$$

Este proceso es relativamente sencillo y es posible implementarlo en el propio hardware de adquisición, permitiendo no sólo reducir a la mitad el total de memoria necesaria sino también el volumen de transacciones de datos entre las diferentes partes del sistema. Es sobre estas señales, una vez han sido alojadas en la memoria de la GPU, sobre las que se aplica la descomposición analítica:

$$\hat{s}_{i,j}[n] \rightarrow \hat{s}_{Ii,j}[n] + j\hat{s}_{Qi,j}[n] \tag{12}$$

Así, introduciendo la interpolación descrita en la ecuación 6, los elementos de fase y cuadratura de la ecuación 9 se expresan respectivamente como:

$$a_I[k,l] = \sum_{i=1}^{N} \sum_{j=i}^{N} b_{i,j} \hat{\S}_{Ii,j}[k,l] \tag{13}$$

$$a_Q[k,l] = \sum_{i=1}^{N} \sum_{j=i}^{N} b_{i,j} \hat{\S}_{Qi,j}[k,l] \tag{14}$$

Donde finalmente tan sólo queda aplicar la ecuación 9 para obtener la imagen deseada. El proceso queda descrito algoritmicamente en pseudo-código en el Algoritmo 1, donde se observa que la duplicidad de la cadena de procesamiento es compensada en parte porque el cálculo de índices es compartido por ambas cadenas.

3. Diseño del conformador de haz en GPU

Para ilustrar el proceso de conformación de haz se ha escogido como escenario una pieza de aluminio de dimensiones $75 \times 150 \times 25$ mm, representada en la Figura 2(a). Este tipo de pieza es una probeta característica en END donde se emplea para calibrar las lentes de focalización en los equipos de imagen. La zona de interés se ha delimitado en rojo en la Figura, y hemos tomado tres tamaños de imagen: $I = N_H \times N_V$, 256×256, 512×512 y 1024×1024 píxeles como referencias para evaluar el costo.

El sistema de adquisición es un equipo de imagen ultrasónica de 128 canales (SITAU 111, Dasel Sistemas). Este sistema proporciona datos con una resolución de 12 bits, con una longitud máxima de señal de 4096 muestras. El transductor es un array lineal de 128 canales con una frecuencia central de $5MHz$ y un ancho de banda del 65 % (PA 8617-B101, Imasonic). Sobre esta plataforma experimental se ha obtenido FMC de $128 \times 128 \times 4096$ datos que servirá de base al desarrollo experimental. Los datos del FMC han sido almacenados con números en precisión simple (4 bytes)[1]. Adicionalmente, y con objeto de poder compararlo con otros resultados de la literatura, se considerará un subconjunto de 64 elementos para componer un FMC de $64 \times 64 \times 4096$ que también será usado en el análisis comparativo.

Como sistema de procesamiento se han tomado distintas plataformas GPU de NVIDA y que se encuentran descritas en la Tabla 1. Podemos observar como cada una de ellas cuenta con un número diferente de cores y con dos tipos de arquitectura: las plataformas #1, #2 y #3 están basadas en la arquitectura Fermi, mientras que las plataformas #4 y #5 están basadas en Kepler (NVIDIA (2014)). El modelo de desarrollo sobre estas plataformas es CUDA (extensión de C/C++ para GPU). Todos las pruebas han sido ejecutadas sobre el sistema operativo Microsoft® Windows 7.

3.1. Pre-Procesamiento

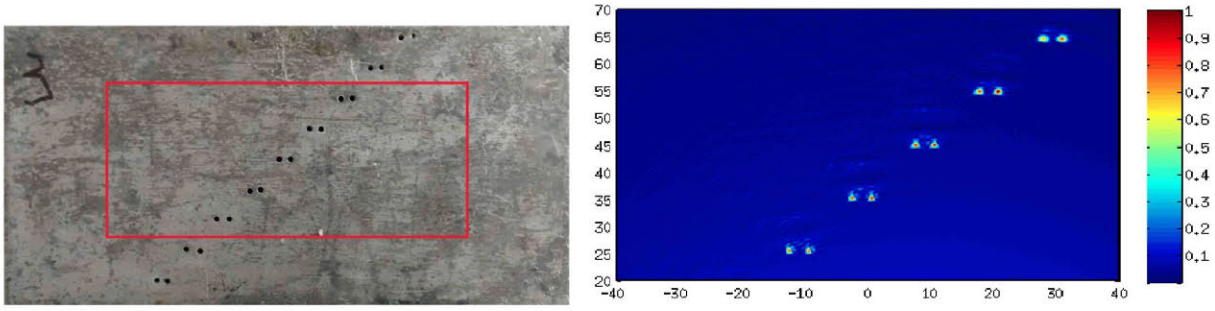
Asumiendo que la reducción por *half matrix* ha sido acometida en el propio hardware de adquisición, con el subsiguiente beneficio de reducir el tráfico de datos entre la plataforma de adquisición y la de procesamiento, el resto de operaciones se desarrolla ya sobre nuestra plataforma objetivo. El cálculo de la señal analítica en la GPU se basa en una implementación paralela de la FFTW (FFTW, 2015) proporcionada por NVIDIA a través de CUDA (NVIDIA, 2014). Además, dado que los datos originales son de 12 bits estos se pueden codificar sobre 16 bits. Es posible emplear unas rutinas matemáticas optimizadas para alta velocidad en las GPU de NVIDIA que, pese a no cumplir con las especificaciones IEEE, según nuestra experiencia son capaces de mejorar el rendimiento en un 150 % sin degradar el resultado final.

La Tabla 2 muestra los tiempos de la descomposición analítica obtenidos para cada una de las plataformas para una longitud

[1] Todo este conjunto de datos está a disposición de los investigadores en la web www.geus.itefi.csic.es

Algoritmo 1 Algoritmo de conformación de haz

Pre-procesamiento
$\hat{s}_{i,j}[n] = s_{i,j}[n] + s_{j,i}[n]$ — Reducción de datos
$\hat{s}_{Ii,j}[n] + j\hat{s}_{Qi,j}[n] \leftarrow \hat{s}_{i,j}[n]$ — Señal analítica
Conformación de haz
for $k = 0$ **to** R_H **do**
  for $l = 0$ **to** R_V **do**
    $a_I = 0, a_Q = 0$ — Inicializar pixel
    for $i = 1$ **to** N **do**
      for $j = i$ **to** N **do**
        $m, \Delta m$ — Calcular índice
        if $0 \leq m \leq L$ **then**
          $\hat{\S}_{Ii,j}[k,l] \leftarrow (1 - \Delta m)\hat{s}_{Ii,j}[m] + \Delta m\hat{s}_{Ii,j}[m+1]$ — Muestra interpolada
          $\hat{\S}_{Qi,j}[k,l] \leftarrow (1 - \Delta m)\hat{s}_{Qi,j}[m] + \Delta m\hat{s}_{Qi,j}[m+1]$ — Muestra interpolada
          $a_I \leftarrow a_I + b_{i,j}\hat{\S}_{Ii,j}[k,l]$ — Multiplica por $b_{i,j}$ y acumula la suma en a_I
          $a_Q \leftarrow a_Q + b_{i,j}\hat{\S}_{Qi,j}[k,l]$ — Multiplica por $b_{i,j}$ y acumula la suma en a_Q
        end if
      end for
    end for
    $A[k,l] \leftarrow \sqrt{a_I^2 + a_Q^2}$ — Cálculo de envolvente
  end for
end for
return A — Imagen final



(a) (b)

Figura 2: Detalle de la pieza de aluminio usada como escenario e imagen ultrasónica obtenida como resultado de aplicar el TFM para la zona señalada

	Modelo	**RAM**	**N^o cores**
#1	GeForce 540M (Fermi Optimus)	GDDR3, 1 Gbytes	96
#2	Quadro 2000 (Fermi)	GDDR5, 1 Gbytes	192
#3	Quadro 4000 (Fermi)	GDDR5, 2 Gbytes	256
#4	Quadro K2000 (Kepler)	GDDR5, 2 Gbytes	384
#5	Quadro K5000 (Kepler)	GDDR5, 6 Gbytes	1536

Tabla 1: Plataformas de procesamiento GPU consideradas

de señal de 4096 muestras. Aunque estos tiempos no son indicativos del rendimiento final, muestran el límite máximo que estas plataformas pueden alcanzar. De este modo vemos como la mejor de las plataformas (#5, Kepler) limita la tasa a 333 imágenes por segundo, mientras que en el peor caso, plataforma #1, este límite queda fijado a 65 imágenes por segundo. En los resultados siguientes el coste de esta operación estará incluido a efectos de evaluación de rendimiento.

	#1	#2	#3	#4	#5
Hilbert(CUFFT)	0.016	0.011	0.006	0.009	0.003

Tabla 2: Tiempos de procesamiento en segundos de la transformada de Hilbert del FMC en las distintas plataformas

3.2. *Desarrollo del conformador en GPU*

El desarrollo en GPU involucra un cambio en el paradigma de programación tradicional debido al modelo SIMD (*Single Instruction Multiple Data*). Además, los distintos tipos de memoria con diferentes prestaciones y tiempos de acceso obligan a analizar con detalle el papel que juegan los datos y parámetros dentro del algoritmo, con el fin de explotar al máximo la capacidad de procesamiento del sistema.

3.2.1. *Implementaciones directas*

La primera idea respecto a la implementación del algoritmo en GPU consistió en lanzar un hilo de ejecución por cada píxel de la imagen, tal y como se muestra en Algoritmo 1. Para llevarlo a cabo se ha definido un grid computacional de tamaño $B_X = \lceil \frac{N_H}{T_{B_X}} \rceil$ y $B_Y = N_V$ bloques de T_{B_X} hilos para lanzar el *kernel*. Este enfoque *naive*, desarrollado en Implementación 1, mantiene todos los resultados parciales en los registros de los procesadores escalares de la tarjeta gráfica, evitando escribir estos cálculos directamente sobre memoria global. Los tiempos de ejecución de esta primera implementación, así como de las siguientes, se analizarán con detalle en la Tabla 3 de la sección de resultados.

El rendimiento de este enfoque se puede mejorar mediante un empleo adecuado de los recursos de la GPU. Una vez identificados los elementos de proceso, acorde a su uso, podemos distribuirlos entre los diferentes tipos de memoria. En general, esta tarea puede ser abordada con diferentes mecanismos: la memoria compartida, que es de tamaño pequeño pero tiene velocidad de acceso rápido ya que se encuentra en el mismo chip que el procesador; la memoria de textura, que es cacheable y se puede utilizar para escribir y leer todos los datos de las operaciones asegurándonos coalescencia entre los datos; y la memoria constante, que es rápida pero sólo se puede utilizar para datos de las operaciones de lectura (NVIDIA, 2014).

Atendiendo a estas consideraciones de memoria, esta primera implementación se puede optimizar como se muestra en la Implementación 2 (GPU1op), donde se ha utilizado la memoria constante para almacenar los valores del filtro espacial y la memoria de textura para almacenar la matriz de datos y las coordenadas espaciales. Esto último proporciona una ventaja adicional gracias a que los mecanismos de textura proporcionan interpolaciones lineales directamente por hardware, de forma que la operación de interpolación ya no es necesaria explícitamente cuando se recupera el valor de la muestra. Esta organización de datos mejora sustancialmente el tiempo de procesamiento, reduciéndolo aproximadamente al menos a la mitad respecto a la implementación *naive* (ver Tabla 3).

Este tipo de implementaciones que corresponde a una solución clásica, ha sido habitualmente desarrollado en sistemas CPU y presenta inconvenientes por la dispersión de las lecturas sobre el banco de muestras lo que baja el rendimiento de forma ostensible (Birk et al., 2011). En GPU, al disponer distintos tipos de memoria y poder distribuir los datos en ellas de acuerdo a su conveniencia, este problema es menos dramático. No obstante el problema persiste y debe ser considerado, por lo que hemos diseñado una tercera implementación.

3.2.2. *Implementación optimizada*

La idea es partir del enfoque planteado en (Nikolov, 2001), que desarrolló una implementación hardware para múltiples FPGAs. que se presenta a la izquierda de la Figura 3. El sistema se basa en la generación de imágenes TFM de baja resolución (*Low Resolution Images*, LRI) compuestas a partir de cada elemento emisor y todos sus receptores. A continuación, sumando las N LRI, se obtiene una imagen final de alta resolución completamente focalizada en todos los puntos de la imagen.

No obstante, debido a la diferencia entre plataformas, trasladar el planteamiento de Nikolov a la GPU involucra cambios sustanciales respecto a lo desarrollado en las implementaciones anteriores. Nuestros requerimientos de memoria se incrementan en N veces y necesitamos desarrollar dos etapas para completar el mismo. Este desarrollo se muestra en la Implementación 3 (GPU2).

La primera etapa, representada por el kernel uno, se ocupa de generar las LRI. Este kernel dedica un hilo por cada píxel de la imagen, y se define un grid computacional tridimensional en el kernel, con $B_X = \lceil \frac{N_H}{T_{B_X}} \rceil$, $B_Y = \lceil \frac{N_V}{T_{B_Y}} \rceil$ y $B_Z = N$ bloques de $T_{B_X} \times T_{B_Y}$ hilos en cada dimensión. Es importante hacer notar que ahora estamos creando bloques 3D y, de esa manera, hay menos bloques por multiprocesador lo que mejora el rendimiento (Figura 3 derecha). El tamaño de bloque debe ser igual al número de elementos de la matriz con el fin de cubrir cada elemento en la emisión. Cada hilo dentro de un bloque se encarga de calcular la suma parcial de cada combinación emisión-recepción. Globalmente este kernel precisa un gran espacio de almacenamiento pero el correcto acotado de éste en bloques limita la dispersión entre lecturas y evita degradar el rendimiento.

Una vez calculadas todas las imágenes de baja resolución (LRI), el segundo kernel se encarga de combinarlas (Implementación 3 kernel 2). Para ello se define en cada hilo un segundo grid con $B_X = \lceil \frac{N_H}{T_{B_X}} \rceil$ y $B_Y = \lceil \frac{N_V}{T_{B_Y}} \rceil$ bloques de $T_{B_X} \times T_{B_Y}$. Este segundo kernel es responsable de calcular la suma de un píxel dado a través de las múltiples imágenes LRI. Con este enfoque los tiempos de computación se reducen drásticamente, como se comprueba en los resultados de la Tabla 3. Dependiendo de la plataforma considerada, esta optimización reduce el coste

Implementación 1 GPU1. Kernel *naive*

$\hat{s}_{Ii,j}[n]$, $\hat{s}_{Qi,j}[n]$ — Señales almacenadas en memoria constante
$x[k,l]$ — Coordenadas especiales almacenadas en memoria constante
$b_{i,j}$ — Filtro espacial almacenado en memoria constante
x_i — Coordenadas del array en memoria constante
$k \leftarrow threadIdx.x + blockIdx.x * blockDim.x$ — Calcular coordenada k en función de índice de hilo
$l \leftarrow blockIdx.y$ — Calcular coordenada l en función del índice de bloque

1: **for** $i = 1$ **to** N **do**
2: **for** $j = i$ **to** N **do**
3: $\hat{m} = (t_j[k,l] + t_i[k,l] - t_0)/\tau_s$ — Cálculo del factor de interpolación
4: **if** $0 \leq m(x,z) \leq L$ **then**
5: $\hat{\S}_{Ii,j} \leftarrow (1 - \Delta m)\hat{s}_{Ii,j}[m] + \Delta m \hat{s}_{Ii,j}[m+1]$ — Muestra interpolada
6: $\hat{\S}_{Qi,j} \leftarrow (1 - \Delta m)\hat{s}_{Qi,j}[m] + \Delta m \hat{s}_{Qi,j}[m+1]$ — Muestra interpolada
7: $a_I \leftarrow a_I + b_{i,j}\hat{\S}_{Ii,j}$
8: $a_Q \leftarrow a_Q + b_{i,j}\hat{\S}_{Qi,j}$
9: **end if**
10: **end for**
11: **end for**
12: $A[k,l] \leftarrow \sqrt{a_I^2 + a_Q^2}$
13: **return** $A[k,l]$

Implementación 2 GPU1op. Kernel Optimización de recursos y memoria

$\hat{s}_{Ii,j}[n]$, $\hat{s}_{Qi,j}[n]$ — Señales almacenadas en memoria de textura
$x[k,l]$ — Coordenadas especiales almacenadas en memoria de textura
$b_{i,j}$ — Filtro espacial almacenado en memoria constante
x_i — Coordenadas del array en memoria compartida
$k \leftarrow threadIdx.x + blockIdx.x * blockDim.x$ — Calcular coordenada k en función de índice de hilo
$l \leftarrow blockIdx.y$ — Calcular coordenada l en función del índice de bloque

1: **for** $i = 1$ **to** N **do**
2: **for** $j = i$ **to** N **do**
3: $\hat{m} = (t_j[k,l] + t_i[k,l] - t_0)/\tau_s$ — Cálculo del factor de interpolación
4: **if** $0 \leq m(x,z) \leq L$ **then**
5: $\hat{\S}_{Ii,j} \leftarrow texture\{\hat{s}_{Ii,j}, \hat{m}\}$ — Interpolación por hardware en GPU
6: $\hat{\S}_{Qi,j} \leftarrow texture\{\hat{s}_{Qi,j}, \hat{m}\}$ — Interpolación por hardware en GPU
7: $a_I \leftarrow a_I + b_{i,j}\hat{\S}_{Ii,j}$
8: $a_Q \leftarrow a_Q + b_{i,j}\hat{\S}_{Qi,j}$
9: **end if**
10: **end for**
11: **end for**
12: $A[k,l] \leftarrow \sqrt{a_I^2 + a_Q^2}$
13: **return** $A[k,l]$

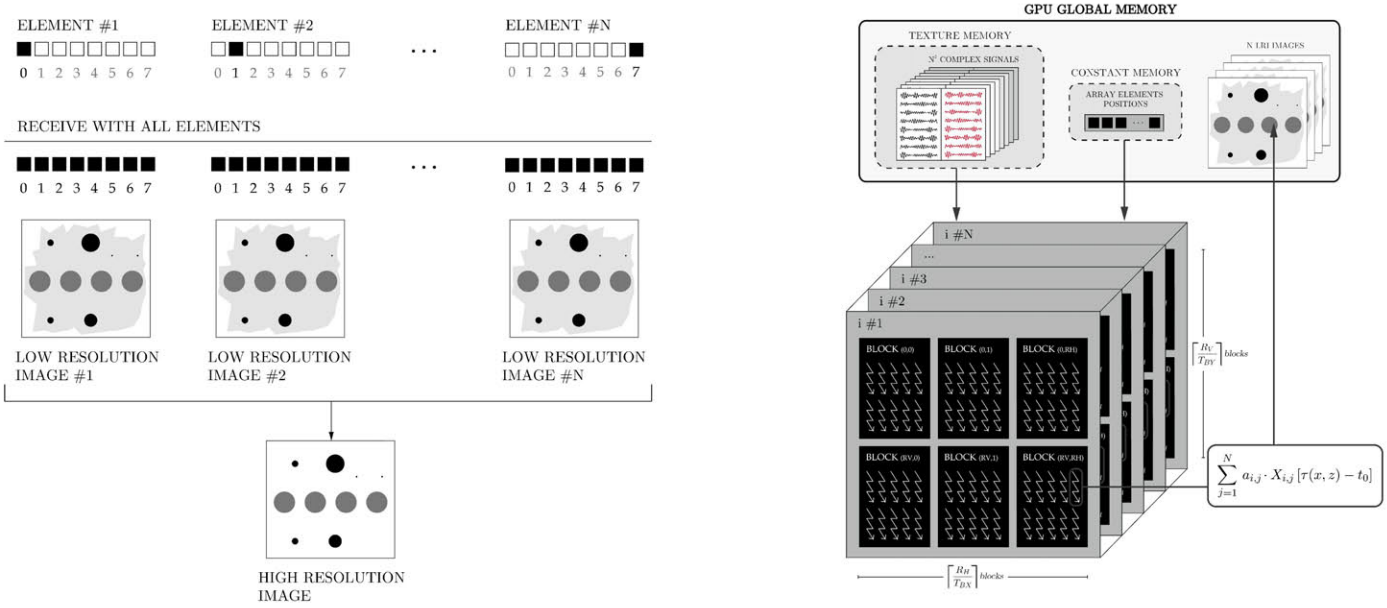


Figura 3: A la izquierda se muestra esquemáticamente la propuesta de Nikolov (2001). Cada combinación de emisor y todos los receptores se utiliza para crear una imagen de baja resolución; posteriormente todas se combinan para crear la imagen final de alta resolución. En nuestra implementación, presentada a la derecha, por cada píxel y elemento emisor lanzamos un hilo de ejecución

computacional entre 2 y 6 veces respecto al algoritmo GPU1op con 64 elementos, y entre 3 y 10 veces si se consideran 128 elementos.

4. Evaluación de rendimiento

La evaluación de las distintas implementaciones fue realizada para las plataformas descritas en la Tabla 1 y los resultados se exponen en la Tabla 3. Con el fin de obtener un valor representativo de las estrategias se han tomado dos modelos de arrays lineales, de 64 y 128 elementos respectivamente, y se han considerado tres tamaños de imagen. Cada resultado es el promedio de 16 ejecuciones. Los tiempos resultantes no incluyen los correspondientes a la adquisición y a la transmisión de datos entre el sistema de adquisición y la CPU, pero sí los tiempos de copia de la memoria RAM de la CPU a la RAM GPU.

Además, el rendimiento de las implementaciones 2 y 3 fue analizado con la herramienta *NVIDIA NSight* que reveló que casi el 40 % del coste computacional es debido a la transferencia de datos desde el sistema de adquisición a la GPU a través de la CPU. Este análisis mostró además, que mientras la implementación 2 precisa de 18 registros nuestra propuesta reduce este número a 12, lo que repercute en la mejora las transacciones con la memoria y permite un mayor grado de ocupación de los *single processors* (un 99.24 % frente al 93.99 % de la implementación 2).

Los resultados muestran claramente la superioridad de la implementación GPU2 en todas las plataformas consideradas y para todos los tamaños de imagen. Además manifiestan una disparidad entre plataformas, que no sólo se debe a los procesadores sino también a la calidad de los recursos auxiliares como las memorias. Así, las mejoras entre la implementación directa GPU1op y GPU2 son relativas a cada plataforma, pudiendo incrementar la velocidad para 64 elementos de 2 (#1) a 6 veces (#5).

Un aspecto interesante es que tanto GPU1 como GPU1op castigan mucho su rendimiento cuando se dobla el número de sensores, multiplicando hasta por cuatro el coste computacional de forma proporcional al número de señales. Esto no sucede así para GPU2, que en todas las implementaciones cuando el número de señales se cuadriplica apenas multiplica el coste por 1.5. En este caso el incremento parece ser independiente de la plataforma, lo que no sucede en las implementaciones directas.

Respecto al aumento de puntos en la imagen, su impacto en el coste de cálculo está por debajo del incremento en el número de puntos, afectando más al algoritmo GPU2 que a las implementaciones de computación directa. De hecho, en la computación directa los incrementos van de 2 a 3 veces, mientras que en los casos de GPU2 este incremento está entre 3.2 y 3.8. Nuestra interpretación es que esto se debe a las altas necesidades de memoria que tiene GPU2 y que se ven fuertemente afectadas en este caso.

Con el fin de valorar mejor el uso del algoritmo del TFM para aplicaciones de tiempo real, las Figuras 4 y 5 muestran la tasa de imagen para arrays de 64 y 128 elementos, respectivamente, considerando la implementación GPU2. De estas gráficas se deduce que cuatro de las plataformas estudiadas son capaces de obtener tasas por encima de 20 imágenes por segundo para un tamaño de 256×256 píxeles. De hecho, dos de ellas (plataformas #3 y #5) alcanzan tasas muy superiores, de 35 y 68 imágenes por segundo respectivamente. Si consideramos el caso de un array de 128 elementos vemos como tres de ellas se sitúan por encima de 15 imágenes por segundo, siendo de nuevo la plataforma #5 la que consigue una tasa de imagen muy alta, de 40 imágenes por segundo para un FMA de $128 \times 128 \times 4096$ señales. No obstante, las imágenes grandes de 1024×1024 están

Implementación 3 GPU2 kernel uno

$\hat{s}_{Ii,j}[t], \hat{s}_{Qi,j}[t], x[k,l]$ — Señales almacenadas en memoria de textura
$b_{i,j}$ — Filtro espacial almacenado en memoria constante
$k \leftarrow threadIdx.x + blockIdx.x * blockDim.x$ — Calcular coordenada k en función de índice de hilo/bloque
$l \leftarrow threadIdx.y + blockIdx.y * blockDim.y$ — Calcular coordenada l en función de índice de hilo/bloque
$i \leftarrow threadIdx.z + blockIdx.z * blockDim.z$ — Calcular coordenada i en función de índice de hilo/bloque

1: $LRI_I[i,k,l] = 0, LRI_Q[i,k,l] = 0$ — Inicialización
2: **for** $j = 1$ **to** N **do**
3: $\hat{m} = (t_j[k,l] + t_i[k,l] - t_0)/\tau_s$ — Calcular factor de interpolación
4: **if** $0 \leq m(x,z) \leq L$ **then**
5: $\hat{\S}_{Ii,j} \leftarrow texture\{\hat{s}_{Ii,j}, \hat{m}\}$ — Interpolación por hardware en GPU
6: $\hat{\S}_{Qi,j} \leftarrow texture\{\hat{s}_{Qi,j}, \hat{m}\}$ — Interpolación por hardware en GPU
7: $LRI_I[i,k,l] \leftarrow LRI_I[i,k,l] + b_{i,j}\hat{\S}_{Ii,j}$ — Almacenar como textura GPU
8: $LRI_Q[i,k,l] \leftarrow LRI_Q[i,k,l] + b_{i,j}\hat{\S}_{Qi,j}$ — Almacenar como textura GPU
9: **end if**
10: **end for**
11: **return** LRI_I, LRI_Q — Imágenes en baja resolución

Implementación 3 GPU2 kernel dos

$k \leftarrow threadIdx.x + blockIdx.x * blockDim.x$ — Calcular coordenada k en función de índice de hilo/bloque
$l \leftarrow threadIdx.y + blockIdx.y * blockDim.y$ — Calcular coordenada l en función de índice de hilo/bloque

1: $a_I[k,l] = 0, a_Q[k,l] = 0$ — Inicialización
2: **for** $i = 1$ **to** N **do**
3: $a_I[k,l] \leftarrow a_I[k,l] + texture\{LR_I[i,k,l]\}$ — Leer desde memoria de textura
4: $a_Q[k,l] \leftarrow a_Q[k,l] + texture\{LR_Q[i,k,l]\}$ — Leer desde memoria de textura
5: **end for**
6: $A[k,l] \leftarrow \sqrt{a_I[k,l]^2 + a_Q[k,l]^2}$
7: **return** A — Imagen final

Tabla 3: Tiempos de cómputo (en segundos) para los tres tipos de implementaciones propuestas con diferentes tamaños de visualización y para arrays de 64 y 128 elementos

Implementaciones		**64 elementos**			**128 elementos**		
		256×256	512×512	1024×1024	256×256	512×512	1024×1024
#1	GPU1	0.369	0.821	2.451	1.423	3.196	9.554
	GPU1op	0.179	0.488	1.604	0.662	1.433	4.140
	GPU2	0.072	0.230	0.851	0.100	0.330	1.275
#2	GPU1	0.343	0.724	2.078	0.918	1.917	5.559
	GPU1op	0.136	0.436	1.232	0.425	0.863	2.435
	GPU2	0.046	0.133	0.479	0.052	0.190	0.689
#3	GPU1	0.325	0.681	1.954	0.948	1.947	5.593
	GPU1op	0.124	0.340	0.974	0.444	0.904	2.528
	GPU2	0.028	0.09	0.355	0.060	0.139	0.537
#4	GPU	0.339	0.694	2.019	1.040	2.089	6.083
	GPU1op	0.119	0.381	1.173	0.484	0.960	2.707
	GPU2	0.042	0.103	0.420	0.072	0.168	0.669
#5	CPU	0.312	0.629	1.732	0.854	1.533	4.190
	GPU1op	0.172	0.316	0.893	0.415	0.736	1.981
	GPU2	0.015	0.038	0.140	0.025	0.060	0.227

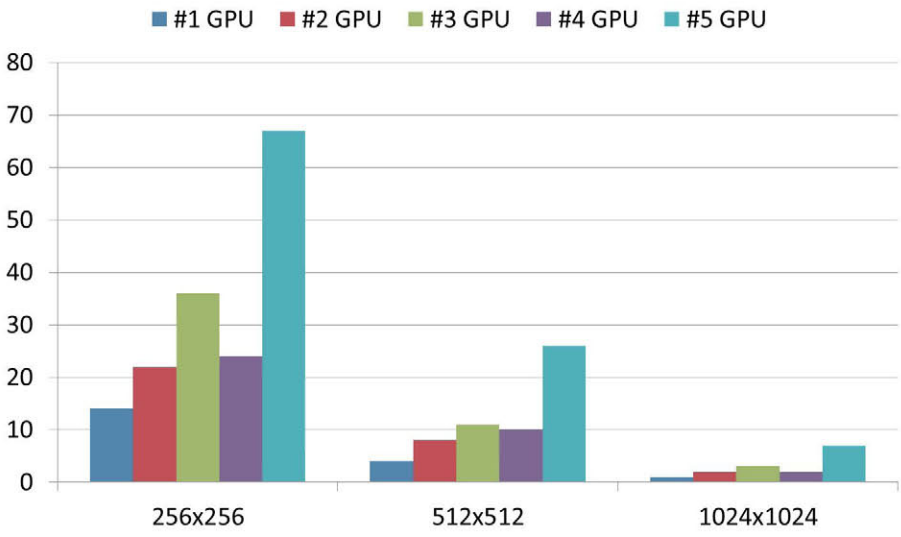


Figura 4: Tasa de imágenes para un array de 64 elementos

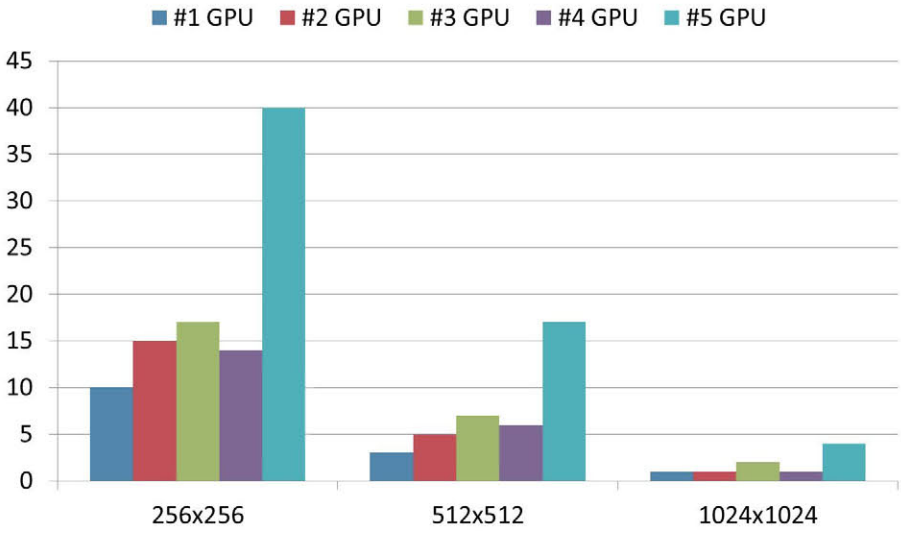


Figura 5: Tasa de imágenes para un array de 128 elementos

aún por debajo de la tasa de 10 imágenes por segundo.

5. Conclusiones

Este trabajo muestra como las actuales tecnologías GPGPU son capaces de realizar procesos de conformación de haz y componer imagen ultrasónica en tiempo real. En este sentido, una solución software constituye una alternativa viable a implementaciones hardware específicas.

Además, se ha realizado un estudio de cómo paralelizar los procesos de conformación de imagen en la arquitectura de la GPU y optimizar el uso de sus recursos. Así, la primera de las propuestas se ha centrado en la implementación más directa del algoritmo, adaptándola al modelo de ejecución SIMD de las tarjetas gráficas. Tras analizar los factores que limitaban su rendimiento, esta primera implementación se optimizó explotando las particularidades de la memoria disponible en el dispositivo acorde al uso de los datos. De esta manera se consiguió reducir aproximadamente a la mitad el tiempo de la conformación de imagen respecto al primer enfoque.

El análisis de esta implementación permitió identificar dos de los cuellos de botella que más influyen en el tiempo de conformación de imagen. Por un lado, la transferencia de memoria entre CPU y GPU, que es una cuestión tecnológica a resolver sobre la que los fabricantes ya están planteando soluciones. Por otro lado la coalescencia de datos, que limita la velocidad de ejecución del algoritmo y puede afectar a la escalabilidad del mismo sobre distintos tamaños de aperturas.

Este segundo problema ha sido tratado mediante el uso de LRI. Esta implementación, además de reducir los recursos internos de la etapa de procesamiento, acota el espacio de lectura de datos de los procesadores. Todo esto permite un mejor uso de los bancos de memoria y mantener más ocupados a los multiprocesadores para disminuir latencias. Como resultado se obtiene un algoritmo más rápido. Así para una imagen de 512×512 píxeles somos capaces de conseguir un sistema en tiempo real de 25 img/seg, y si se consideran imágenes de 256×256 hasta 65 img/seg.

Los resultados son prometedores pues demuestran que la implementación de un conformador SAFT de imagen ultrasónica TFM+FMC es posible mediante un desarrollo software con un equipo de gama media-alta. Con este enfoque es posible simplificar el equipo de ultrasónico a un sistema de adquisición multicanal menos complejo y costoso que los equipos convencionales. Por último, es previsible que el desarrollo tecnológico de estas plataformas habilitará el empleo de otras técnicas de conformación más complejas en sistemas de tiempo real.

English Summary

Analysis of a software implementation of an ultrasonic signal beamformer in real-time

Abstract

This paper studies the software implementation in an ultrasonic imaging system of Total Focusing Method. In order to accomplish real-time requirements parallel programming techniques have been used. Then, using GPGPU techniques, two different implementation alternatives are analysed, showing how proper planning of access to data improves the performance of the algorithm.

Keywords:

Ultrasonic imaging GPU Signal Processing Parallel computing CUDA

Agradecimientos

Este trabajo ha sido desarrollado bajo los proyectos DPI2010-19376 y FIS2013-46829R.

Referencias

Birk, M., Guth, A., Zapf, M., Balzer, M., Ruiter, N., Hübner, M., Becker, J., 2011. Acceleration of image reconstruction in 3D ultrasound computer tomography: An evaluation of CPU, GPU and FPGA computing. In: Conference on Design and Architectures for Signal and Image Processing (DASIP). Tampere, pp. 1–8.

Camacho, J., 2010. Imagen ultrasónica por coherencia de fase. Ph.D. thesis, Facultad de Ciencias Físicas.

Camacho, J., Martinez, O., Parrilla, M., Mateos, R., Fritsch, C., Oct 2007. A strict-time distributed architecture for digital beamforming of ultrasound signals. In: Intelligent Signal Processing, 2007. WISP 2007. IEEE International Symposium on. pp. 1–6.

FFTW, 2015. Library website.
URL `www.fftw.org`

Hansen, J. M., Hemmsen, M. C., Jensen, J. r. A., Mar. 2011. An object-oriented multi-threaded software beamformation toolbox. In: D'hooge, J., Doyley, M. M. (Eds.), SPIE Medical Imaging: Ultrasonic Imaging, Tomography, and Therapy. pp. 79680Y–79680Y–9.

Holmes, C., Bruce W. Drinkwater, Wilcox, P. D., Dec. 2005. Post-processing of the full matrix of ultrasonic transmit-receive array data for non-destructive evaluation. NDT & E International 38 (8), 701–711.

Holmes, C., Drinkwater, B. W., Wilcox, P. D., Nov. 2008. Advanced post-processing for scanned ultrasonic arrays: application to defect detection and classification in non-destructive evaluation. Ultrasonics 48 (6-7), 636–42.

Hunter, A. J., Drinkwater, B. W., Wilcox, P. D., Nov. 2008. The wavenumber algorithm for full-matrix imaging using an ultrasonic array. IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control 55 (11), 2450–62.

Jensen, J., Holm, O., Jerisen, L., Bendsen, H., Nikolov, S., Tomov, B., Munk, P., Hansen, M., Salomonsen, K., Hansen, J., Gormsen, K., Pedersen, H., Gammelmark, K., May 2005. Ultrasound research scanner for real-time synthetic aperture data acquisition. Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on 52 (5), 881–891.

Kortbek, J., Nikolov, S. I., Jensen, J. r. A., Mar. 2007. Effective and versatile software beamformation toolbox Jacob. In: Emelianov, S. Y., McAleavey, S. A. (Eds.), Medical Imaging 2007: Ultrasonic Imaging and Signal Processing.

M2M, 2015. Gekko: Advanced phased-array ut.
URL `http://www.m2m-ndt.com/products/Gekko_features.htm`

Martín-Arguedas, C. J., 2010. Técnicas de apertura sintética para la generación de imagen ultrasónica. Ph.D. thesis, Universidad de Alcalá.

Nikolov, S. I., 2001. Synthetic aperture tissue and flow ultrasound imaging. Ph.D. thesis, Technical University of Denmark.

Nilsen, C.-I. C., Hafizovic, I., Apr. 2009. Digital beamforming using a GPU. In: IEEE International Conference on Acoustics, Speech and Signal Processing. Ieee, pp. 609–612.

NVIDIA, 2014. CUDA C Programming Guide 6.0. No. February 2014.
URL `www.nvidia.com`

Oppenheim, A. V., Schafer, W. R., Schafer, R. W., Buck, J. R., 1989. Discrete-Time Signal Processing. Vol. 23. Prentice-Hall, Upper Saddle River, New Jersey.

Parrilla, M., 2004. Conformación de haces ultrasónicos mediante muestreo selectivo con codificación delta. Ph.D. thesis, Universidad Politecnica de Madrid.

Romero-Laorden, D., Martínez-Graullera, O., Martín-Arguedas, C. J., Ibáñez, A., Gómez-Ullate, L., Apr. 2012. Paralelización de los procesos de conformación de haz para imagen ultrasónica con técnicas GPGPU. Revista Iberoamericana de Automática e Informática Industrial 9 (2), 144–151.
URL http://linkinghub.elsevier.com/retrieve/pii/S1697791212000039http://www.sciencedirect.com/science/article/pii/S1697791212000039

Romero-Laorden, D., Martínez-Graullera, O., Martín-Arguedas, C. J., Pérez-Lopez, M., Gómez-Ullate, L., 2011. Paralelización de los procesos de conformación de haz para la implementación del Total Focusing Method. In: 12 Congreso Español de END. Valencia.

Romero-Laorden, D., Martínez-Graullera, O., Martín-Arguedas, C. J., Tokio Higuti, R., Octavio, A., 2009. Using GPUs for beamforming acceleration on SAFT imaging. In: IEEE International Ultrasonics Symposium. IEEE, Rome, Italy, pp. 1334–1337.

Rougeron, G., Lambert, J., Iakovleva, E., Lacassagne, L., Dominguez, N., 2013. Implementation of a GPU Accelerated Total Focusing Reconstruction Method within CIVA Software. 40th Annual Review of Progress in Quantitative Nondestructive Evaluation 1581 (1), 1983–1990.

Siritan, T., Techavipoo, U., Worasawate, D., Keinprasit, R., Pinunsottikul, P., Sugino, N., Thajchayapong, P., 2013. Beamforming Complexity Reduction Methods for Low-Cost FPGA-based Implementation. In: Biomedical Engineering International Conference (BMEiCON-2013). pp. 2–5.

So, H. K. H., Chen, J., Yiu, B. Y. S., Yu, A. C. H., 2011. Medical Ultrasound Imaging: To GPU or not to GPU. IEEE Micro 31 (5), 54–65.

Sutcliffe, M., Weston, M., Dutton, B., Charlton, P., Donne, K., 2012. Real-time full matrix capture for ultrasonic non-destructive testing with acceleration of post-processing through graphic hardware. NDT & E International 51, 16–23.

Szabo, T. L., 2004. Diagnostic Ultrasound Imaging. Elsevier.

Wall, K., Lockwood, G. R., 2005. Modern Implementation of a Realtime 3D Beamformer and Scan Converter System. In: IEEE International Ultrasonics Symposium. Vol. 00. pp. 1400–1403.

Wang, L., Shi, D., Zhao, A., Tan, C., Liu, D. C., May 2011. Real-Time Scan Conversion for Ultrasound Imaging Based on CUDA with Direct3D Display. In: Conference on Bioinformatics and Biomedical Engineering (iCBBE). Ieee, pp. 1–4.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5780361

Zhang, F., Bilas, A., Dhanantwari, A., Plataniotis, K. N., Abiprojo, R., Stergiopoulos, S., 2002. Parallelization and Performance of 3D Ultrasound Imaging Beamforming Algorithms on Modern Clusters. In: Proceedings of the 16th international conference on Supercomputing. pp. 294–304.